Metrics and Tool for System Assessment*

G. Bucci, F. Fioravanti, P. Nesi, S. Perlini
Department of Systems and Informatics, University of Florence
Via di S. Marta 3, 50139, Florence, Italy
[bucci@ingfi1.ing, fioravan@dsi, nesi@ingfi1.ing, perlini@dsi].unifi.it

Abstract

A wide increment of the object-oriented diffusion has been registered. This is accompanied by the needs of metrics and tools for assessing class reusability, maintainability, complexity, etc. Metrics have to produce confident results and have to be continuously revalidated on the basis of specific purpose on their adoption. Tools for system assessment must be capable of satisfying the needs of each company by supporting the definition of more specific metrics and by providing appropriate views of software characteristics. These views are useful to give at the developers as well as at the managers immediately understandable representations of the system status. In this paper, a new approach is proposed for system assessment. This approach is based on the adoption of new and well-known metrics together with a set of histograms and profiles that give a clear characterization of the system under development. The semantics of these histograms has been validated against several projects. A brief overview of the tool, TAC++, developed for system assessment has been also included.

Index terms: *object-oriented metrics, system analysis, assessment tool, profiles and histograms.*

1 Introduction

In the recent years, the Object-Oriented Paradigm (OOP) has been adopted for its promises about portability, reusability, maintainability, etc. Obviously, these features are not automatically achieved by adopting the OOP, even if an object-oriented methodologies for development are used. The adoption of the OOP implies changes in the whole development process (i.e., project management, resource evaluation, resource planning, assessing, documenting, requirement analysis, analysis, design, coding, testing, etc.) [10]. Due the needs of certification and standardization, project management and assessment are becoming key issues in the processes of development and maintenance. In order to guarantee the control of the development process, quantitative metrics for evaluating and predicting product characteristics must be used. Product features are typically: quality (see ISO 9126), cost, reuse, conformance with the requirements and the requests of the market, etc., and company goals for each specific product. The description of product features is typically considered the product profile.

A growing attention to the process of software development has created the need to get process-oriented information, both real data and measured features have to be continuously compared for controlling the process and, when needed, for adjusting the model (continuous process identification and improvement). This process of adaptation is usually performed by adjusting weights and thresholds [6] till the development process is identified, then it is maintained under control. The evaluation model can be based on technical and cognitive metrics for the indirect measure of: (i) effort of development, (ii) effort to understand for reuse, (iii) effort to understand for maintenance.

Some studies with metrics and measurement frameworks for object-oriented systems have been presented in [7], [11], [12], [5], where general concepts for the estimation of system size, complexity and reuse level have been proposed together with many other metrics. Unfortunately, the effort in defining new metrics has not been supported by the implementation of assessment tools. Presently, the commercial assessment tools are only capable of (i) estimating a limited number of metrics (some present a tool for defining new metrics), (ii) visualizing specific views (profile). These are not supported by an integrated tool for evaluating weights and tuning metrics for the different project profiles and development life-cycle phases.

Therefore, an integrated framework for developing and maintaining under control the system under development must be supported by tools for (i) defining and evaluating direct and indirect metrics, (ii) defining and showing suitable views of the system (profiles, diagrams, tables, graphs, histograms, etc.) as well as of its components/classes (views should be focussed on assessing quality, conformance to the OOP, etc.), (iii) tuning metrics by estimating weights and scale factors, (iv) controlling project evolution by using reference/threshold values, (v) collecting and comparing projects trends (the basis for the continuos improvement).

Please note that different views, profiles, histograms of the same entity/class can be required by different people involved in the project (project and task managers, developers, etc.), or by the same people in different phases of the development life-cycle. These views must be focussed on highlighting different aspects. Measured values must also be compared against minimum, maximum and typical values and trends defined for the development phase under observation on the basis of the product profile required. These in turn have to be defined by the company considering its experience.

This paper reports a set of cognitive metrics (metrics which include the measure of the aspects related to the comprehension/understanding of the program/class/system) and their interpretation for the assessment of C++ applications. The metrics proposed belong to a framework and tool specifically defined for C++ language, the research prototype TAC++ (Tool for Analyzing C++ code). TAC++ is comprised of an integrated class browser/editor, which is capable of estimating the values of several direct metrics. On these bases, high-level indirect metrics can be obtained.

The main aspects of the metric framework have been inherited from these presented in [11], [12]. Due to the high number of metrics, which are available in the framework (including the most diffuse and well–know metrics in the literature), only few of them have been discussed, some other metrics have been presented in [12], [2]. The metrics proposed can be used to produce histograms which can be very useful for analyzing the conditions/suitability of the system under assessment. Moreover, these histograms can be very useful for detecting critical conditions – e.g., identifying classes with bad design and class hierarchy with unsuitable organization, and poor comprehensibility. The semantics of the histograms and the suitability of the metrics discussed have been validated on the basis of several small-medium sized projects.

2 A Selection of Metrics for Cognitive Assessment

In this section, a selection of metrics for cognitive assessment is presented. The selection has been performed on the basis of the authors experience – e.g., [11], [12] and considering the several similar experiences presented in the literature – e.g. [8]. According to these experiences, the processes of reuse, maintenance, development with libraries, prediction of costs, etc. can be aided by the adoption of cognitive measures. These in turn can be concretized by using direct an indirect measures; such as cognitive complexity, class complexity/size, the structure of the inheritance class tree, the cohesion/coupling among methods, the cohesion/coupling among classes, etc. [7], [8], [6], [11].

The metrics reported in the next subsections can be classified in *Method-*, *Class-* and *System-Level* metrics. These metrics can be profitably used for system assessment highlighting some of the above mentioned aspects. The system assessment consists in the analysis of the values of metrics with respect to the quantity of methods or classes having those values for the whole system. In this way, histograms of the system with respect to the selected metrics are obtained. Qualitatively, these histograms are only marginally dependent on the language and on the application field. Similar histograms have been presented in [8] and [1], by considering other metrics. For most of the metrics discussed a validation process has been performed on the basis of several projects by using a multilinear regression analysis [14].

In this paper, some new metrics and a further level of analysis for the above mentioned histograms are introduced. It will be shown that it is very useful to analyze the trend of the histogram to detect possible anomalies and disfunctions.

To help the reader to understand the metric formulation and discussion, the authors have prepared Tab.4 in which the metrics and their corresponding meaning are reported in alphabetic order.

2.1 Method-Level Metrics

For evaluating complexity/size of methods, most of the traditional mainly functional metrics (such as the number of Lines Of Code (LOC), the Halstead measure (Ha) [4], the McCabe complexity (Mc) [9]), can be profitable used. By using these metrics, data flow aspects related to method parameters are neglected [15] (the set of parameters of a method represent a cognitive measure of its complexity). In order to avoid this problem, more general metrics including the external interface of methods have been defined.

In general, complex as well as big methods are difficult to be understood and, thus, to be maintained and reused. For this reason, the evaluation of method complexity/size can be useful for measuring system understandability. Thus, in order to guarantee methods reusability, a maximum acceptable value for method complexity/size has to be respected. This maximum value must be set on the basis of company needs and experiences – e.g., by evaluating the cost of maintenance of each method in terms of effort for a couple of years in several projects.

Bounds for LOC of class methods have been identified on the basis of the research group and industrial experiences and projects in order to maintain the costs of modification low – i.e., a mean of 20 LOC per method. For the maximum value of LOC per method, two distinct cases have been recorded on the basis of different types of application; in particular, 150 LOC for systems endowed with a Graphic

User Interface (GUI) and about 50 for systems without GUI. These values are very similar to those reported in [8].

2.2 Class Level Metrics

By using Mc, Ha, and LOC metrics it is possible to define a set of pure functional class metrics (the sum of complexities or size of all class methods). It has been often demonstrated that such metrics are not very suitable for evaluating object-oriented projects, since they are not capable of considering the object-oriented aspects [13], [5], [12]. In fact, they neglect information about class specialization (*is_a*, that means code and structure reuse), and class association and aggregation (*is_part_of* and *is_referred_by*, that mean class/system structure definition and dynamic managing of object sets, respectively). On the other hand, WMC (Weighted Methods for Class) [1], and LOC (used in [8]) have been adopted as good compromises between precision and simplicity of evaluation.

2.2.1 Class Complexity/Size.

In [3], a fully object-oriented metric for evaluating class complexity/size has been presented together with a comparison with the above mentioned object-oriented metrics. It includes cognitive, structural and functional aspects. The class complexity/size, CC, has been defined as the sum of the External Class Description (ECD) (complexity/size due to class definition and method interface definition) and the Internal Class Implementation (ICI) (method implementation):

$$CC = ECD + ICI, \tag{1}$$

where CC components can be decomposed in complexities due to locally and inherited class members:

$$ECD = ECDL + ECDI, \tag{2}$$
$$ICI = w_{CL}CL + w_{CI}CI, \tag{3}$$

where:

$$\begin{aligned} ECDL &= w_{CACL}CACL + w_{CMICL}CMICL, \\ ECDI &= w_{CACI}CACI + w_{CMICI}CMICI, \end{aligned} \tag{4}$$

and thus: $CACL$ is the Class Attribute Complexity Local (complexity due to attributes locally defined); $CACI$ is the Class Attribute Complexity Inherited; $CMICL$, the Class Method Interface Complexity Local (complexity of local method parameters); $CMICI$, the Class Method Interface Complexity Inherited (as the previous for methods inherited); CL, the Class complexity/size due to Local methods; and CI, the Class complexity/size due to Inherited methods. Metrics CL and CI measure the complexity/size of the functional part estimated by using one of the above mentioned metrics: Ha, Mc, LOC. (Please note that CL evaluated with Mc is equal to WMC.)

Thus, according to the metric used, CC may be considered a complexity or a size metric integrating cognitive aspects. Metric CC takes into account both structural (attributes, relationships of *is_part_of* and *is_referred_by*) and cognitive (methods, method "cohesion" by means of $CMICL$ and $CMICI$, respectively), and functional by means of CL, CI, aspects of the class.

Weights in the above metrics have to be evaluated by using of a multilinear regression [14] on the basis of actual class effort. The weight values obviously depend on the purpose and on the phase in which the metric is evaluated. Thus, a trend for the weights along the development and/or maintenance and/or the reuse process has to be determined.

It has been demonstrated [12] that these metrics produce slightly better results regarding to correlation and variance with respect to metrics: WMC [1], to that in [13], and that in [5] for the same purpose. Therefore, the evaluation of CC, as well as that of other simple class complexity/size metrics (e.g., WMC, CI), is very useful for maintaining under control the evolution of system classes, and thus of the whole system. The control of the indiscriminate growing of classes is operated by defining suitable thresholds according to the company experience and goals, as well as to the OOP. The identification of these thresholds is performed by using the same process adopted for evaluating those of method-level metrics. Similar experiences have led some authors to declare these reference values on the basis of their experience [8], [3], etc.

The above mentioned ECD metric gives a measure of what can be observed by reading the class definition, for example in the phase of class reuse or maintenance. It is a measure of the cognitive complexity of the class. Even more useful is the analysis of $ECDL$ and $ECDI$ for evaluating whether the class or its superclasses are sufficiently easy to be understood, and thus to be reused or maintained at reasonable costs. Moreover, ECD can be used for predicting class complexity/size during system analysis/early-design.

In Tab.1, the reference values obtained for CC- and ECD-based metrics are reported. CC has been evaluated by considering the weights evaluated in [12]. Thus, the results obtained for CC cannot be compared with those produced by the other metrics. When maximum values were reached and overcome, a decrease in understandability appears with a corresponding decrease in maintainability and reusability.

In the table the minimum values are also reported since each class has to provide a minimum ECD to be understandable. Peaks for $ECDI$ and $ECDL$ are typically

	Mean Values	Max Values	Min Values
CC	200	1500	–
ECD	350	1500	20
$ECDI$	200	2000	80
$ECDL$	150	1600	20
ICI	200	1500	-
CI	150	1200	-
CL	50	700	-

Table 1. Reference values for CC- and ECD-based of metrics.

present in 100 and 30, respectively.

2.2.2 Number of Class Attributes and Methods

A simple approach to class size evaluation can be based on counting the number of local attributes and methods (see metric $Size2 = NAL + NML$ defined by Li and Henry in [7], sum of the number of local attributes and methods). On the other hand, this simple counting of class members could be in many cases too coarse. For example, when an attribute is an instance of a very complex class its presence in a class often implies a high cost of method development which is not considered simply by increasing NAL of one unit. Moreover, $Size2$ does not consider the class members inherited (that is, reuse). For these reasons, in order to improve the metric precision, a more general metric has been defined by considering the sum of the number of class attributes and methods both local defined and inherited, respectively:

$$\begin{aligned} NAM &= w_{NAL}NAL + w_{NML}NML \\ &\quad + w_{NAI}NAI + w_{NMI}NMI. \end{aligned} \tag{5}$$

Also in this case, the typical values of weights can be estimated by using a multilinear regression technique. If the purpose is to detect critical conditions (of excessive cost of reuse or development) the weights can be imposed to be equal to 1. Please note that, these metrics can be used as indirect measures (i) since the early phases of software development for predicting class size and thus class development costs, (ii) on libraries for evaluating the effort of adoption/reuse.

These metrics are strongly dependent on the role of the class. GUI classes usually present a higher number of attributes and methods. According to company experience as in [8], [3] suitable thresholds have been evaluated and can be imposed in order to control the indiscriminate growing of classes in terms of attributes and methods. Similar suggestions, for constraining these numbers, have been also suggested by several other authors. Differently from the other metrics discussed the evaluation of these metrics is really cheap.

In Tab.2 the reference values are shown. As discussed, a strong difference between normal and GUI classes (on left and right of each column, respectively) was detected. When the maximum value is reached and overcome an increase of the reuse and maintenance cost have been registered due to the decrease of understandability which in turn is due to the increment of general complexity.

	Mean Values	Maximum Values
NAM	45 - 117	69 - 189
$NAMI$	30 - 78	46 - 126
$NAML$	15 - 39	23 - 63
NA	9 - 27	15 - 45
NAI	6 - 18	10 - 30
NAL	3 - 9	5 - 15
NM	36 - 90	44 - 144
NMI	24 - 60	36 - 96
NML	12 - 30	18 - 48

Table 2. Reference values for NAM family of metrics.

Please note that the number of local methods tends to be bigger in C++ rather than in other languages since C++ needs of a mean of at least 2 constructors and one destructor.

2.2.3 Deep Inheritance Tree and the Number of Superclasses.

The structure of the inheritance hierarchy impacts on system maintainability, reusability, extensibility, testability, etc. since a high number of superclasses can make classes hard to understand and test. In the literature, the so-called DIT, Depth of Inheritance Tree metric has been proposed. DIT estimates the number of direct superclasses until the root is reached [1]. It ranges from 0 to N (0 in the case of root classes). Metric DIT is strongly correlated with maintenance costs [7].

Initially, no rules were given for solving multiple inheritance cases. In the implementations reported in the literature, for the case of multiple inheritance, DIT metric evaluates the maximum value among the possible paths towards the several roots (for example 5 for class M, see Fig.1) or the mean value among the possible values 3, 4, and 5, thus 4. These measures are an over-simplification of the real conditions – e.g., class M presents 7 superclasses and, thus, its complexity obviously depends on all these classes. For this reason, metric $NSUP$, Number of SUPerclasses, has been defined by the authors. Also in this case, suitable bounds must be defined for controlling the object-"orientedness" of the hierarchy and the related features.

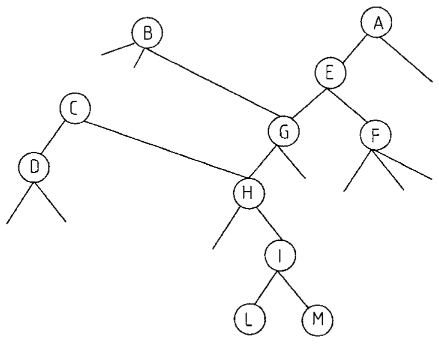

Figure 1. A specialization hierarchy with multiple inheritance.

2.2.4 Number of Children and the Number of Subclasses.

In order to better analyze the position and the relevance of a class in the class hierarchy, it is very important to evaluate the number of its direct subclasses. To this end, the so-called NOC, metric has been defined. This estimates the Number Of Children, considering only the direct subclasses [1]. It ranges from 0 to N (0 in the case of a leaf). Classes with a high number of children must be carefully designed and tested since their code is used by several entities. On the other hand, a high number of children can be due to a lack of hierarchyzation. Thus, this metric can be useful for identifying critical classes. This metric is also strongly correlated with maintenance costs as demonstrated in [7].

The same punctualization made for DIT can be made for NOC. According to the example of Fig.1, class E has two children without considering that it presents at least 6 subclasses (considering only those reported in the figure). For this reason, $NSUB$, Number of SUBclasses, has been defined. It counts the subclasses until leaves are reached. Thus $NSUB$ can be regarded as a more precise measure of class relevance in the system. Also in this case, suitable bounds have to be defined.

2.2.5 DIT, NSUP, NOC and NSUB, Reference Values.

In Tab.3, the reference values for the inheritance tree analysis metrics are shown. A strong difference between normal and GUI classes has been detected, but only regarding the maximum values. When the maximum value of $NSUP$ is overcome the class inherits too much features to be easily understandable. On the other hand, when the maximum value of $NSUB$ is overcome the implementation of the class is really critical since an error in its body can produce a fault in a wide number of other classes.

In order to identify the max number of SUP for a hierarchy is useful to think about real world. OOP recommend to model classes as close as possible to real object descriptions. Real world rarely need more than 5 nested levels for completely describe the target object.

	Mean Values	Maximum Values
DIT	2	4 - 5
$NSUP$	2	5 - 6
NOC	2	10 - 20
$NSUB$	5	30 - 90

Table 3. Reference values for metrics on class hierarchy.

According to our experiments $NSUP$ and $NSUB$ are better ranked for detecting critical conditions with respect to DIT and NOC; in the sense that they are more sensitive to the specialization mechanisms.

2.3 System-Level Metrics

Several system-level metrics have been presented in order to evaluate the conformity of the project to the OOP and the above mentioned features [7], [8], [6], [11], such as: (i) mean method complexity; (ii) mean number of methods per class; (iii) mean number of attributes per class; (iv) mean number of classes per tree (typically C++); (v) maximum depth of inheritance, (vi) maximum number of classes in a tree, (vii) number of trees, etc. These values give a concise system assessment, but in several occasions they can lead to wrong deductions since the manager (the evaluator) is not capable of identifying the critical conditions.

To solve this problem, several system histograms have been defined and adopted. The histograms report the system status by depicting the behavior of the above mentioned metrics with respect to the quantity of classes and/or methods. For example: (i) Number of Methods as function of method complexity/size; (ii) NCL, Number of CLasses, as a function of CC; (iii) NCL as a function of Number of Attributes (NA); (iv) NCL as a function of Number of Methods (NM), etc.

As shown in the next Section, the mentioned histograms are much more expressive than the single evaluation of a simple mean value of a class metric, since they describe the behavior of the metric in the context of the system – e.g., the value of a class-level metric for all classes. This behavior can be compared with typical behaviors registered for similar projects in the same company. For example, mean and maximum values can be satisfactory, but may exist in the system one or more classes with a complexity greater than the maximum value. In addition, the qualitative shape of the histogram is independent of the application context and language.

It is also very useful to maintain under control the evolution of these histograms for analyzing and, thus, for detecting problems of the development and/or maintenance process.

3 A System Assessment

In this section, the adoption of the mentioned system histograms and bounds is shown by using a small-sized project (LIOO, Lectern Interactive Object Oriented). It consists in a class framework for modeling music implemented under UNIX operating system by using calls to Xwindows. In its development a special attention to reuse and maintainability aspects have been given. It was comprised of 113 classes in its first version and 136 in the fifth, LIOO5; several intermediate versions were analyzed. In LIOO5 the number of classes was close to the final number, but some classes remained to be completed yet. This has been experienced by observing LIOO6 and the final version, LIOO7. The class framework has been totally reused in the ESPRIT HPCN project MOODS (Music Object Oriented Distributed System).

In general, it has been observed that it is marginally relevant to have the system compliant with simple bounds since the system may present suitable mean values even with critical conditions. For this reason, the histograms corresponding to the metrics discussed must be analyzed in order to identify conditions to be corrected. The resolution of the critical conditions strongly reduces the fault probability and the costs of reuse.

3.1 Class Complexity/Size

In Fig.2, the histograms of internal class complexity due both to inherited and local methods of LIOO5, are reported. These histograms are very useful to identify classes for which the complexity is too high with respect to the acceptable costs of analyzability and/or further development. A correct histogram has to present a low metric value at the peak with a rapidly decreasing trend for higher values. By observing CI histogram, several classes with unacceptable values bigger than 1200 were detected. This can be due to (i) a too deep class hierarchy, or (ii) the presence of strongly complex superclasses, or (iii) the presence of GUI-based classes. In the case of LIOO, some classes for modeling the elementary features of the GUI were built. For the CI histogram, 27 classes with $CI = 0$ are also present. These can be (i) structures (in C++ structures are considered classes as well) or (ii) root classes or (iii) stand-alone classes (which are also roots). The verification of these conditions is quite easy. Otherwise, classes with $CI = 0$ are too abstract, and thus have to be furtherly analyzed to verify the justification of the lack of complexity. This can be due to (i) a disputable definition of the hierarchy or to (ii) the presence of the system in an early phase of development.

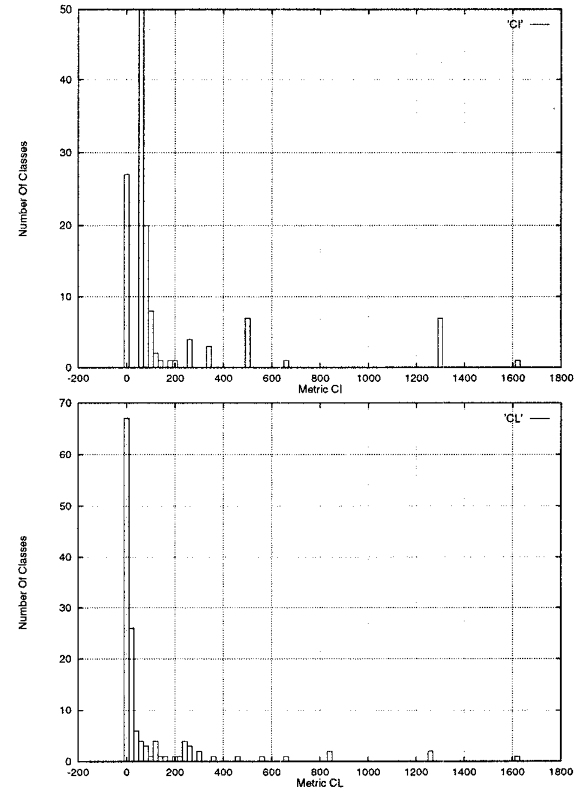

Figure 2. Histograms for CI and CL.

Observing the results obtained for CL, few classes with values bigger than 700 were detected. Their complexity was due to their unjustified growing. In some cases, the development leads to implement methods that could be more correctly placed in their superclasses. Therefore, it is suggested to control these classes in order to verify if their methods can be moved in the superclass (maybe identifying duplications of methods in subclasses). Please note that, for CL histogram, there exists several classes with CL close to 0. These are classes which optimally exploit the class hierarchy, while classes with $CL = 0$ are typically structures or classed under construction.

3.2 External Class Description

In Fig.3, $ECDI$ and $ECDL$ histograms are reported, for LIOO5. The ECD gives an indirect measure of understandability of the class by observing its attributes and method interfaces. The $ECDI$ histogram presents a peak in 100 satisfying the criteria stated in the previous Section. It also evident the presence of 27 classes with $ECDI$ close to zero. These are the already identified root classes, stand alone classes and structures. $ECDL$ histogram presents a peak which contains all the classes with metric value between 0 and 100; most of these classes assume values in the range $20-40$, according to minimum value presented in the previous section. Please note that there exists two classes with extremely high values for $ECDL$. From the analysis of LIOO5, it results that a great part of the complexity for these classes is due to the attributes. The comparison between the two histograms allows to identify if the system present structural problems (at the level of class hierarchy) or only problems in the leaf classes. If the class hierarchy is not well-structured classes with high $ECDI$ appear. Therefore, in the situation depicted in Fig.3 problems are present only if the leaf classes. These classes have to be inspected in order to verify if their complexity is justified.

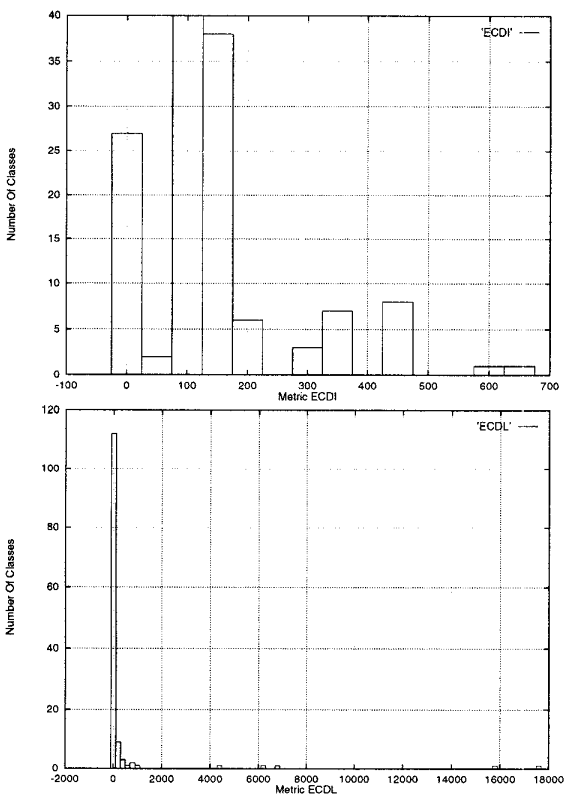

Figure 3. Histograms for $ECDI$ and $ECDL$.

3.3 Number of Class Attributes

In Fig.4, the histograms for the number of attributes, inherited and local, have been reported. In general, when the number of attributes increases, also the number of methods to manage the attributes usually augments. Therefore, in order to maintain limited the class complexity and thus to obtain acceptable values of maintainability and testability for the whole system it is necessary to maintain the number of attributes reasonably low.

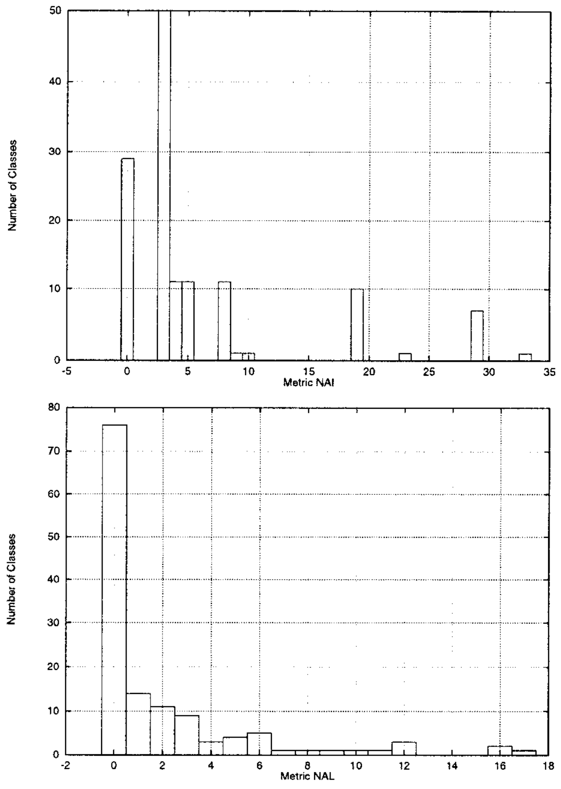

Figure 4. Number of classes per Number of Attributes per class: NAI inherited, NAL local

Please note that in LIOO5, NAL assumes a mean value that accomplishes that reported in Tab.2, while a few classes are outside of the optimal value. Classes with $NAL = 0$ present a mainly behavioral specialization. Simple relationships among NAL, NAI can be supposed in terms of $NSUP$.

In Fig.4, the mean value of NAI is close to 6, but two peaks exist in the profile (at 19 and 29). By eliminating the peaks, a mean value of about 3 is obtained. Therefore, it has been recommended to investigate classes related to peaks since they present too much attributes. After an inspection, it has been observed that these classes have been derived from GUI classes with a large number of protected attributes and so they are obviously out of the nominal range for NON-GUI classes. In this case, the peaks have to be considered a justified exception.

Classes having a great number of inherited attributes are, in general, derived from classes with a large number of local attributes; or classes very deep in the hierarchy. A comparison between $NSUP$ and NAI, for classes out of the optimal range can help to identify the classes to be investigated. If a class is not so deep in the hierarchy, upper level classes must be investigated in order to discover if all the attributes defined are related to the child classes or if one or more of the parent classes can be split in order to reduce the complexity. In Fig.5 an example of class splitting is reported; classes are represented with ellipses and lower letter are locally defined attributes. By the splitting of the class named B into two classes (generating abstract class $A1$), the number of inherited attributes of leaf classes named C and D is lowered. This operation can be performed only if the attributes can be reassigned to different classes according to their adoption by the class methods. This can be a strong support to make more reusable the class hierarchy.

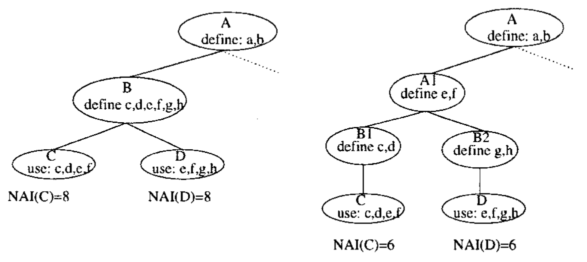

Figure 5. Class splitting example.

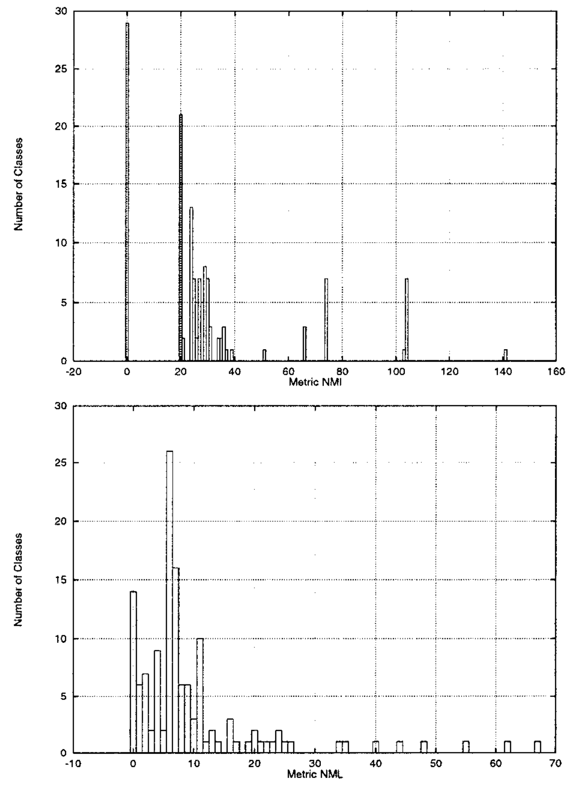

Figure 6. Number of classes per Number of Methods per class: NMI inherited, NML local

3.4 Number of Class Methods

In Fig.6, the number of local and inherited methods per class is reported with respect to the number of classes. These metrics are used for evaluating the functional complexity of the class. In general, a large number of methods denote a high number of functionalities and, thus, the class may result to be too complex. This could be due to a lack of a correct specialization or for the lack of the exploitation of the mechanisms of delegation.

The optimal figure for NML is a graph with a peak close to 7, according to Tab.2, for NON-GUI classes and about three times this value for GUI classes, with a rapid decreasing profile. The classes that have no local methods ($NML = 0$) can be structures or classes that specialize all the methods of the parent classes. A great value for NML may be due to the presence of a lot of virtual methods, having obviously a low functional complexity.

Regarding to NMI, according to Tab.2, a good mean value for NMI can be estimated around 14, with a rapidly decreasing profile. The classes that have $NMI = 0$ are root, stand alone classes or structures, but it is possible to add to this group also classes that specialize all the methods inherited from parent classes.

In Fig.6, considering the inherited part, two peaks out of the optimal profile, are present. In general, classes with this kind of characterization are GUI classes or classes derived from parents having a large number of local methods. Analyzing the data, a mean value of 29 methods for each class has been estimated. It decreases to 24 with the elimination of the two above described peaks.

If classes with a large number of local methods are identified, in order to reduce the collection of too much functionalities in a single class, it is recommended to verify the possibility of splitting these classes in two or more classes specializing the original ones.

3.5 Number of Super- and Sub-Classes

In Fig.7 the figure of $NSUP$ is reported. This metric allows the identification of the number of levels in the class hierarchy. When a lot of classes have more than 6 levels it is recommended to check if it is possible to aggregate more classes in one, in order to reduce the levels. These

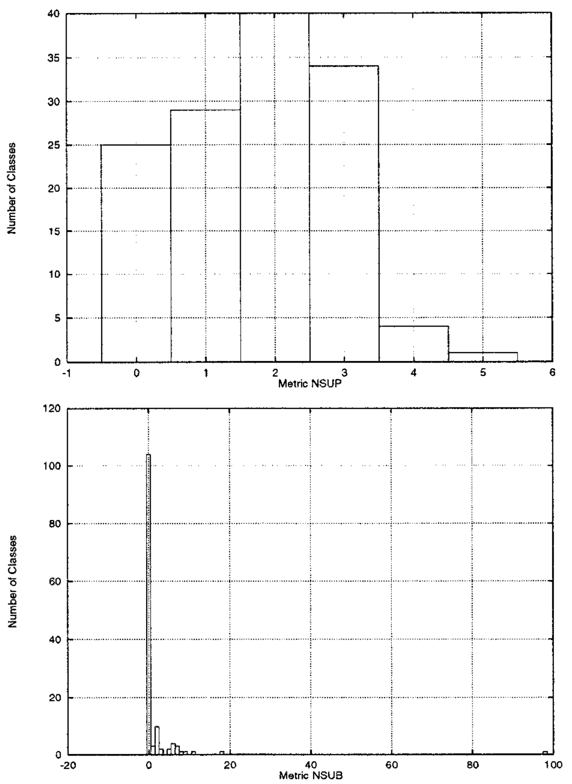

Figure 7. Number of classes per NSUP (superclasses) and NSUB (subclasses).

classes present often unnecessary specialization in the parent classes along the class hierarchy. A high number of level in the class hierarchy leads to high costs of understanding since the methods used in the leaf classes is distributed among several classes.

For LIOO5, a peak for $NSUP$ equal to 2 or 3 has been obtained. The analysis of the related histogram allows the identification of the number of roots or stand-alone classes, which are classes with $NSUP = 0$. When 6 or more nesting levels appear, it is necessary to re-analyze the class hierarchy in order to verify if such a high number of levels is needed. In Fig.7, it can be noted that LIOO5 has a class with a high number of children (about 100). This class is a root class of a large class hierarchy that cannot be split since it provides the fundamental methods and attributes for graphics.

4 An Overview of TAC++ Tool

The above-mentioned metrics, as many others, can be evaluated by using TAC++ (see Fig.8). TAC++ is a research prototype suitable for studying system (i) development, (ii) maintenance, and (iii) architecture. As can be observed in Fig.8, TAC++ is comprised of six main components addressing the problems of: (i) navigating in the system classes, (ii) evaluating low-level metrics, (iii) defining and evaluating high-level metrics, (iv) defining and showing metric histograms and profiles, (v) statistically analyzing system for validating and tuning metrics, and (vi) collecting real data.

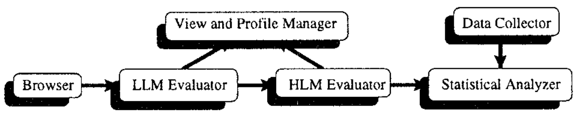

Figure 8. TAC++ Structure, Organization and Features

4.1 Collecting Low-Level Measures

The process of collecting low-level measures is based on a first phase of preprocessing, such as those usually adopted in compilers. The relationships among classes (*is-a*, *is-part-of*, *is-referred-by*, coupling between classes by means of methods, etc.) are identified. If the code of the used class library is available, it can be considered into the assessment process, however, the definitions (C++ headers files) of the used libraries must be available.

The class browser provides an integrated editor for navigating among system classes and their members. The browser shows the list of classes with a synthetic description of their relationship: the class hierarchy, the list of methods, etc. if the method is a Constructor, Destructor or $v$irtual definition from parents (methods), first Virtual definition (methods).

TAC++ is capable of producing Low-Level Metrics (LLM). These are direct metrics – such as LOC, Mc, Ha, NA, NM, NCL, $NAML$, Mc, Ha, etc. The results produced by the LLMs Evaluator in different phases of the software life cycle or system assessment can be collected and analyzed. Please note that, most of the above mentioned direct metrics can produce draft results even when only the class definition is available, such as in the early analysis or when a library is analyzed.

High-Level Metrics (HLMs) can be defined by the user on the basis of LLMs and HLMs. To this end, a specific interactive tool allows the definition of new metrics by means of a visual editor.

4.2 Immediate Visualization of Results

In order to provide a fast and understandable view of the project status, the values obtained for LLMs and HLMs can be visualized in a set of specific views. These views are typically directly defined in the internal quality manual of the company together with many other rules that must be followed for developing the system.

A profile is a consumptive view, which is capable of showing the values of several metrics with respect to their specific typical, maximum, minimum values. The minimum/maximum value(s) can be considered the lowest/greatest value(s) under/over which a correction should be made.

Some examples of typical views are: (i) a view reporting the effort prediction of a class: $Size2$, NAM, WMC, etc.; (ii) a view presenting effort for reuse estimation of a class: CC_m, WMC, Mc, etc.; (iii) views reporting conformity to OOP at system level (general system assessment): NRC (Number of Root Classes of the System), NRC/NCL, mean value of CC, etc.; (iv) views reporting class metrics related to reusability and maintainability: NOC, $NSUP$, $NSUB$, DIT, etc.

By using TAC++, it is also possible to show histograms for each metrics and for all classes of the system under assessment. Several examples of histograms have been already presented in the previous section.

4.3 Validating and Tuning Metrics

Metric validation process consists in identifying metric parameters (weights) and the relevance of metrics components on the basis of the knowledge of actual data, depending on the goals of the metric under validation – e.g., on the basis of the real effort for development, maintenance, or the number of defects encountered, etc.

The validation process can be used for (i) verifying which terms of each identified metric is relevant for its estimation (this can be used for reducing the estimation effort and, in some cases, for increasing correlation and reliability), (ii) evaluating the confidence of the measures obtained, (iii) tuning metric models according to different context and profiles (weights and scale identification), (iv) identifying metric parameters along the development life-cycle for evaluating the development progress with respect to reference trend, and finally (v) for evaluating the goodness of metrics in representing the selected feature.

Usually, the validation is performed by using mathematical and statistical techniques. The engine of Statistic Analyzer used in TAC++ is mainly based on Progress [14], for which a graphical user interface has been added. The Statistic Analyzer is capable of estimating all the metric weights used in a metrics if the corresponding real values are available.

Weight values depend on the application context and, thus, it is possible to obtain more precise results by estimating the weights depending on the type of the system under assessment. This can be simply done by using a little number of reference projects into the selected area and estimating weights with the previously applied method. The reference projects must comply with OOP and quality profiles requested for the project area.

Usually a metric may present a high number of components but not all the terms have the same importance. By using the Statistic tool, it is possible to verify not only the correlation of the whole metric with respect to the real data, but also the correlation of each term of the metric with respect to the collected effort, maintenance or other real data, in general. Thus, the most relevant metric components for the estimation of the targeted features can be identified.

Therefore, real data reporting direct measures of the features that should be evaluated by metrics are needed – e.g., real effort for developing, real effort for maintaining, real effort for the class comprehension, number of defects, etc. This information is collected by using Data Collector, which has been developed in Java to be portable in a wide number of platforms.

5 Conclusions

The adoption of the OOP has produced a great demand of specific metrics. In this paper, several direct and indirect metrics for the evaluation of effort, maintenance and reusability costs, have been introduced together with an approach for their application during system assessment. Results about the statistical evaluation of bounds and typical histograms have been reported. These values were obtained on the basis of several projects and an example of their application in analyzing a medium system has been reported. It has been shown that, by interpreting the histograms produced through the application of these metrics, critical conditions of the whole system can be identified. More specifically, histograms make evident degenerative conditions and allow the identification of classes that require correction (to be easily reused or maintained) or further development. In order to manage the large number of metrics, an integrated tool for defining, showing and validating them is mandatory. TAC++ is a highly configurable environment to control all the aspects of C++ projects since the early phase of system development. TAC++ offers many features for aiding project development and maintenance. It has been profitably used for controlling the development and maintenance of several projects carried out by the research group and in some industry in order to validate its use on the application field. The results obtained by the above mentioned projects have permitted to establish bounds, reference profiles and histograms for a wide typology of applications, that are in use in present and will be used in future C++ projects.

ACKNOWLEDGEMENTS

The authors would like to thank the members of TOOMS, ICOOMM, QV/MOOVI, LIOO and MOODS teams that have accepted to collect data. A special thank to B. Pages for an early version of the class browser (i.e., Xcoral), and A. Borri of CESVIT, for his help in designing the first version of TAC++.

References

[1] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.

[2] F. Fioravanti, P. Nesi, and S. Perlini, "A tool for process and product assessment of c++ applications," in *Proc. of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, (Florence Italy), March 1998.

[3] F. Fioravanti, P. Nesi, and S. Perlini, "Assessment od System Evolution Through Characterization," in *Proc. of Internation Conference on Software Engineering, (Kyoto JP), April 1998.*

[4] H. M. Halstead, *Elements of Software Science*. Elsevier North Holland, 1977.

[5] B. Henderson-Sellers, "Some metrics for object-oriented software engineering," in *Proc. of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 6 Pacific 1991*, pp. 131–139, TOOLS USA, 1991.

[6] B. Henderson-Sellers, D. Tegarden, and D. Monarchi, "Metrics and project management support for an object-oriented software development," in *Tutorial Notes TM2, TOOLS Europe'94, International Conference on Technology of Object-Oriented Languages and Systems*, (Versailles, France), 7-10 March 1994.

[7] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of Systems Software*, vol. 23, pp. 111–122, 1993.

[8] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics, A Practical Guide*. New Jersey: PTR Prentice Hall, 1994.

[9] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[10] P. Nesi, "Managing OO Project Better," *IEEE Software*, vol. in press, 1998.

metric	comment
$CACI$	Class Attribute Complexity/size Inherited
$CACL$	Class Attribute Complexity/size Local
CC	Class Complexity/size
CI	Class Method complexity/size Inherited
CL	Class Method complexity/size Local
$CMICI$	Class Method Interface Complexity/size Inherited
$CMICL$	Class Method Interface Complexity/size Local
DIT	Depth Inheritance Tree [1]
ECD	External Class Description
$ECDI$	External Class Description Inherited
$ECDL$	External Class Description Local
Ha	Halstead metric [4]
$HSCC$	Class Complexity by [5]
ICI	Internal Class Implementation
LOC	number of Lines Of Code
Mc	McCabe ciclomatic Complexity
NOC	Number Of Children [1]
$NSUB$	Number of SUBclasses
$NSUP$	Number of SUPerclasses
NA	Number of Attributes of a class
NAI	Number of Attributes Inherited of a class
NAL	Number of Attributes Locally defined of a class
NM	Number of Methods of a class
NMI	Number of Methods Inherited of a class
NML	Number of Methods Local of a class
NAM	Number of Attributes and Methods of a class
NCL	Number of CLasses in the system
NRC	Number of Root Classes in the system class tree
$Size2$	Number of class attributes and methods [7]
$TJCC$	Class Complexity by [13]
WMC	Weighted Methods for Class [1]

Table 4. Glossary of the metrics mentioned in this paper.

[11] P. Nesi and M. Campanai, "Metric framework for object-oriented real-time systems specification languages," *The Journal of Systems and Software*, vol. 34, pp. 43–65, 1996.

[12] P. Nesi and T. Querci, "Effort estimation and prediction of object-oriented systems," *The Journal of Systems and Software*, vol. in press, 1998.

[13] D. Thomas and I. Jacobson, "Managing object-oriented software engineering," in *Tutorial Note, TOOLS'89, International Conference on Technology of Object-Oriented Languages and Systems*, (Paris, France), p. 52, 13-15 Nov. 1989.

[14] P. J. Rousseeuw and A. M. Leroy, *Robust Regression and Outlier Detection*. New York, USA: Jhon Wiley & Sons, 1987.

[15] H. Zuse, *Software Complexity: measures and methods*. Berlin, New-York: Walter de Cruyter, 1991.